BLG 336E - Analysis of Algorithms II

Project-1 Report

Can Yılmaz Altıniğne
150130132

14 / 03 / 2017

You can compile the program by using ‘g++ 150130132.cpp -std=c++11 -o project1’ command. Since I used unordered_set and *to_string( )* function, I needed to use C++11 standarts.

- **How does your algorithm work? Write your pseudo-code.**

BFS( ):

```
Initialize state_list
Initialize layer_counter = 0, cycle_prevented = 0
Initialize set of visited states                        // unordered_set

Push the initial state to visited states and initial list
Push the initial list to state_list

while state_list[layer_counter] is not empty:

    Initialize a empty list and push it to state_list

    For all states S in state_list[layer_counter]:          //    O(V)

        If S leads to win situation:
            return S

        For all blocks B in S:                    //    Θ(Adj[S])

            If B can move up or left:
                Initialize a new state N with updated block position
                and layer number and layer order of S. // parent

                If N is not visited before:    //    O(1) Average Time
                    add N to visited states
                    add N to state_list[layer_counter+1]

                Else
                    Increment cycle_prevented

            If B can move down or right:
                Initialize a new state N with updated block position
                and layer number and layer order of S.

                If N is not visited before:    //    O(1) Average Time
                    add N to visited states
                    add N to state_list[layer_counter+1]

                Else
                    Increment cycle_prevented

    Increment layer_counter

return dummy_state          //    Not found
```

DFS( ):

```
Initialize state_list                  // For finding parent
Initialize stack of states
Initialize set of visited states       // unordered_set
Initialize cycle_prevented as 0.

Push the initial state to states
Push the initial state to state_list with order 0.

while states is not empty:                                //    O(V)

    Pop the state S from states

    If S is not visited before:               //    O(1) Average Time
        add S to visited states

        If S leads to win situation:
            return S

        For all blocks B in S:                           //    Θ(Adj[S])

            If B can move up or left:
                Initialize a new state N with updated block position
                and order of S in state_list           // parent

                Push the N to states
                Push the N to state_list

            If B can move down or right:
                Initialize a new state N with updated block position
                and order of S in state_list           // parent

                Push the N to states
                Push the N to state_list

    Else
        Increment cycle_prevented

return dummy_state          //    Not found
```

For BFS and DFS, I used the pseudocodes in the course book (*Algorithm Design* by Jon Kleinberg and Éva Tardos). Adding new states to list takes O(E) time. Since I add new nodes when I create them and that can be seen edge between new node and parent node. Because we check every edge once and every vertex once the time complexity is equal for both algorithms and it is O(V+E).

- **What is the extra complexity that is caused by the cycle search?**

Let's consider the cycle check time. I used unordered set for cycle check. It is a hash table, so the average time of accessing an element in that table is O(1). Since I can check that if I visited that state before in constant time, cycle check does not increase the complexity. But if I consider that I get worst case every time (it is highly unlikely), checking cycle has O(V) complexity in worst case and algorithms have O(V + E*V) complexity since I check cycle when I consider every edge.

- **Explain your classes and your methods. What are their purposes?**

  - ***Block Class***

    In this class, I have the properties of blocks in the environment. Bottom left coordinates, orientation, number of block as it is written in the order in input file and height of the block are kept in this class. There are generic get and set methods in the class and also three important methods. The move_down_or_right( ) method increment the bottom left y coordinate of the block, if it is a horizontal block it increments the bottom left x coordinate. The move_up_or_left( ) method decrement the bottom left y coordinate of the block, if it is a horizontal block it decrements the bottom left x coordinate. The *stringify( )* method returns the bottom left coordinates, height and the orientation of the block as string (i.e. 3 5 2 h).

  - ***State Class***

    In this class, I have the properties of states. This class has members for keeping parent state's layer number and order in that layer. For DFS, I just used one layer to keep parents, since when I pop the states from stack I lose them. So the layer number for DFS is 0 for every states and state order for DFS is kept in order_for_dfs variable. Additionally there is a vector for keeping blocks in that state and a 6x6 matrix for visualizing the grid to decide the moves. There are generic get and set methods for the class. The *prepare_the_grid( )* method is used for filling the *visualize_grid* matrix with block numbers if there is not a block in an area of that grid, matrix element is filled with 0. The *check_down_or_right( )* and *check_up_or_left( )* methods take integer value as a parameter. This parameter represents an index for state_blocks vector and these functions look if a move is possible for that block in state_blocks vector. It returns a boolean value. The *stringify( )* method is returns the visualize_grid matrix values in a string. Then this value is used for unordered_set to check cycles. This value is added to unordered_set and check operation is made on these values. The *win_the_game( )* method checks from the rightmost of the block which we want to take out from maze to the exit point if it sees a value other than 0 in that path it returns 0 else it returns 1. The *state_output_generator( )* returns the state representation as string to print it in output file.

- *Algorithms Class*

  This class basically keeps Breadth-First-Search and Depth-First-Search algorithms and variables which are used in implementation (vector, stack). The BFS( ) and DFS( ) methods run the implicit BFS and DFS and they print the moves on output file and other information on standart output.

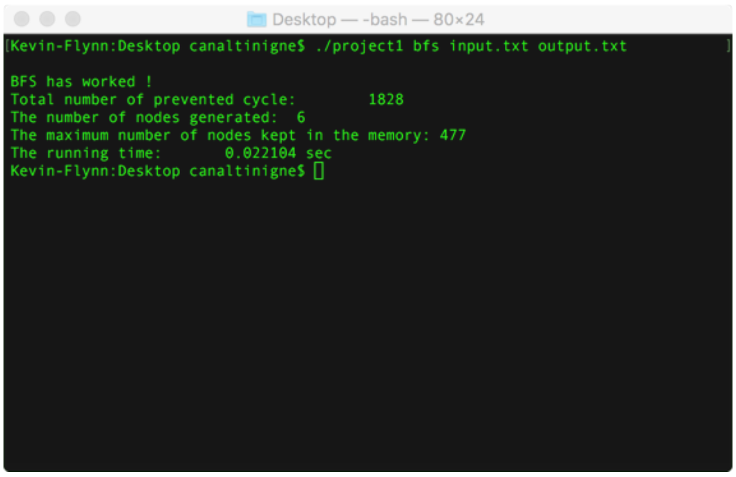
• If you use adjacency list representation, how does the complexity of your algorithm change?

Adjacency list can be used if I know all the states and edges at the beginning without creating the states dynamically as it is done for this homework. Then for the structure of checking the edges would not change, since the total consideration time of edges results in O(E).

In the second case if I do not know all the states and edges between them as it is in this homework. Then, when I create the state I add it to adjacency list. In this homework adjacency list can represent a list of added vertices and every vertex in that list keeps a list of its neighbors. That would not change the adding and extracting a vertex in stack or graph. But when we check if there is a cycle situation every time in every stack/queue operation we need to look at this adjacency list to decide if we have that vertex in that graph with its neighbors list. If we have it then we would not add that node to stack/ queue to prevent cycle. So searching a node in a list takes at most O(N) in this case O(V) time. We check cycle every time we look at edges, checking all edges takes O(E) time for both algorithms. Then total complexity has O(V + E*V) complexity. This complexity is equal to my algorithms' worst cases (very unlikely). The worst case for my implementations is the checking unordered_list could take O(N) time.

• Analyzing the Results

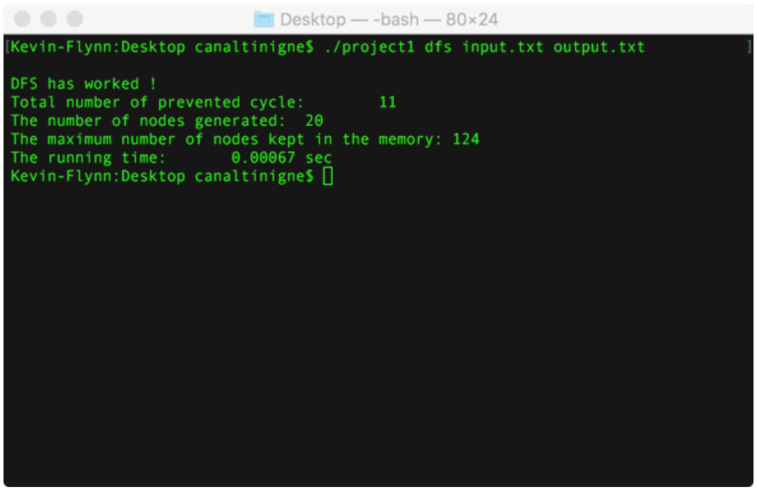
- *BFS*



```
[Kevin-Flynn:Desktop canaltinigne$ ./project1 bfs input.txt output.txt         ]

BFS has worked !
Total number of prevented cycle:        1828
The number of nodes generated:  6
The maximum number of nodes kept in the memory: 477
The running time:       0.022104 sec
Kevin-Flynn:Desktop canaltinigne$ 
```

- ***DFS***



```
[Kevin-Flynn:Desktop canaltinigne$ ./project1 dfs input.txt output.txt          ]

DFS has worked !
Total number of prevented cycle:        11
The number of nodes generated:  20
The maximum number of nodes kept in the memory: 124
The running time:       0.00067 sec
Kevin-Flynn:Desktop canaltinigne$ 
```

- ***Comparison***

The total number of prevented cycle is huge for BFS with respect to DFS. Because in DFS we branch in a path first but for BFS we proceed layer by layer. If we move block up in ancestor vertex, for the child we move block down and for grandchild when we consider moving block up again we conflict with the grandparent so cycle preventation happens. This kind of situations happen a lot of times in BFS but for DFS when we start moving block in a direction we move until we come to borders of grid. As a result total number of prevented cycle in DFS is much less than it is in BFS.

The number of nodes generated for BFS is less than it is in DFS. Since we search layer by layer in BFS if there is a path to reach exit state in K moves we can see that exit state in Kth layer. Branching factor affects the DFS operations so we can miss some moves to reach exit state easily because of the branch we proceed in DFS.

Maximum number of nodes kept in the memory for BFS is more than DFS. Because we keep nodes layer by layer and those layers are getting larger as we proceed.

The running time of DFS is lower than BFS. This actually depends on the structure of graph. In BFS, I traverse all nodes in a layer. If this layer becomes very big, then searching is wasting a lot of time. But in DFS since we consider relatively less nodes if the branch that we proceed results in an exit state, then DFS reach the solution in less time. Also since cycle preventation takes more time in BFS than it does in DFS, the running time takes more time in BFS than DFS but they both have same complexity.